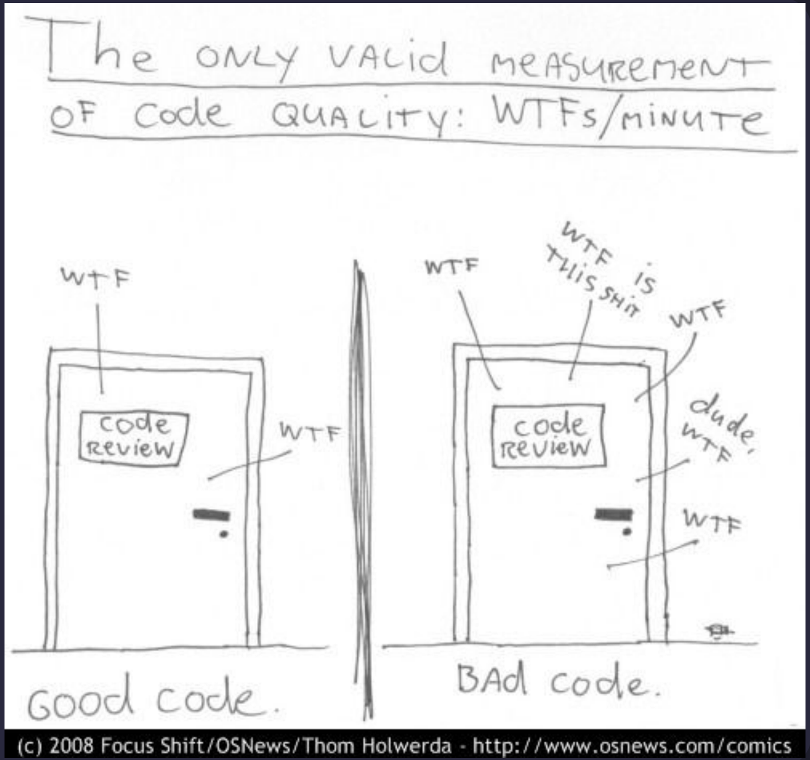
YOUR LOGO HERE

Clean Code Workshop


The only valid measurement of code quality: WTFs/minute
WTF
code review
WTF
Good code.
WTF
WTF is this shit
WTF
code review
dude, WTF
WTF
BAd code.
(c) 2008 Focus Shift/OSNews/Thom Holwerda - http://www.osnews.com/comics

INDEX.HTML

What is clean code?

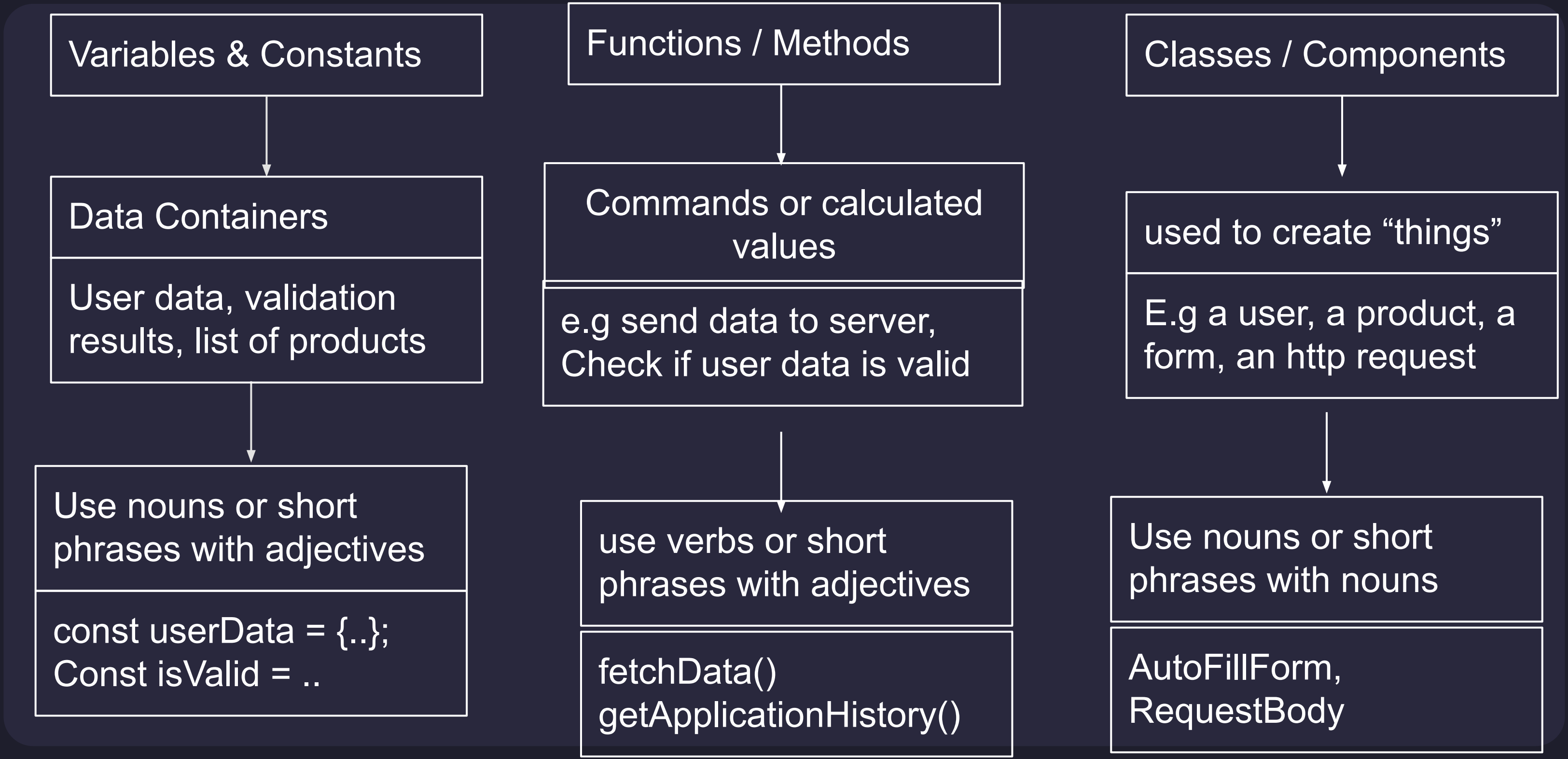
Clean code is a reader-focused development style that produces software that’s easy to write, read and maintain.

Clean code is not a one time action - it is written by iterating and improving over time

MAIN PRINCIPLES:

1. Naming
2. Comments & Formatting
3. Functions
4. Control Structures & Errors

Naming Rules



Variables & Constants

Data Containers

User data, validation results, list of products

Use nouns or short phrases with adjectives

```
const userData = {..};
Const isValid = ..
```

Functions / Methods

Commands or calculated values

e.g send data to server, Check if user data is valid

use verbs or short phrases with adjectives

```
fetchData()
getApplicationHistory()
```

Classes / Components

used to create “things”

E.g a user, a product, a form, an http request

Use nouns or short phrases with nouns

AutoFillForm, RequestBody

Try to avoid “magic” numbers/strings - put as a global variable with a proper name

Naming task

https://github.com/TzachiGitHub/cleanCodeWorkshop/tasks/main/namingTask.js

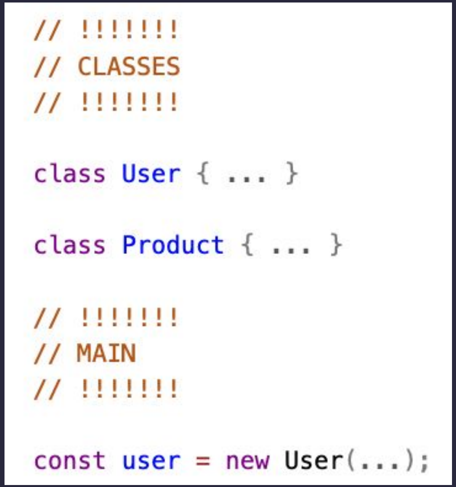
Comments & Formatting

Coding mythbuster - comments are actually bad for code readability!

Bad comments are usually divided into a few sections -

Dividers & Markers:

They are redundant, If your code is written In a clean way -
it is obvious what your Different code parts are about

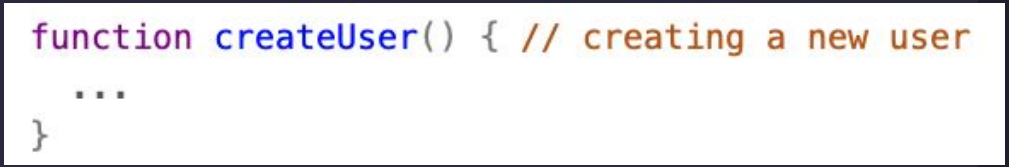


```
// !!!!!!!
// CLASSES
// !!!!!!!

class User { ... }

class Product { ... }

// !!!!!!!
// MAIN
// !!!!!!!

const user = new User(...);
```

Comments & Formatting - bad comments

Redundant Information:

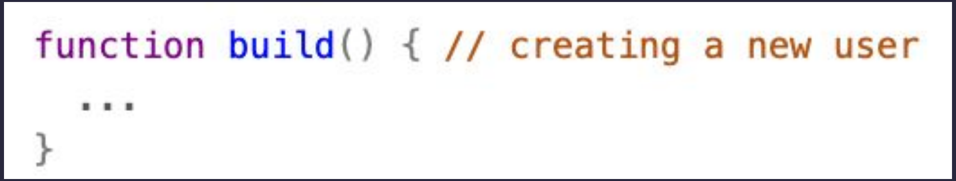
It’s very obvious that createUser is creating a user, there’s no need to comment about it



```
function createUser() { // creating a new user
  ...
}
```

* the comment would be useful if you had bad naming -

But of course we’re avoiding
These Bad names
in the first place

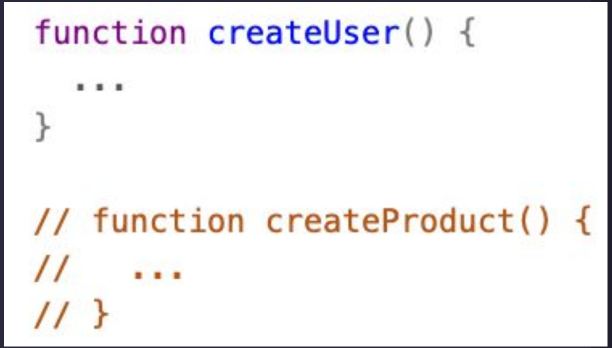


```
function build() { // creating a new user
  ...
}
```

Comments & Formatting - bad comments

Commented Out Code:

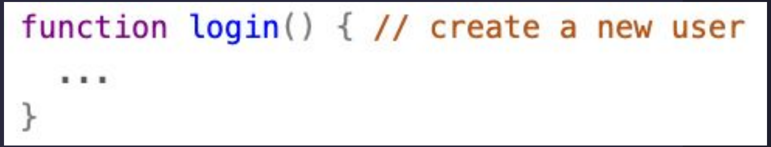
Since we're using git for source control -
We don't need to fear losing code -
we can delete it instead of commenting it out,
and in case there's a need for it again we can
go back previous versions in a click of a button



```
function createUser() {
  ...
}

// function createProduct() {
//   ...
// }
```

Misleading Comments:

Probably the worst kind of comments



```
function login() { // create a new user
  ...
}
```

Const filteredPatients = filterPatients()   // get all patients

Comments & Formatting - GOOD comments

1. Legal Information - in some projects/companies we might be required to add legal information (smart contracts, etc…)

```
// (c) Academind GmbH
```

2. “Requires” Explanation - in rare cases adding extra explanation next to your code does help - even if you wrote the code perfectly.
A good example for it is regular expressions -

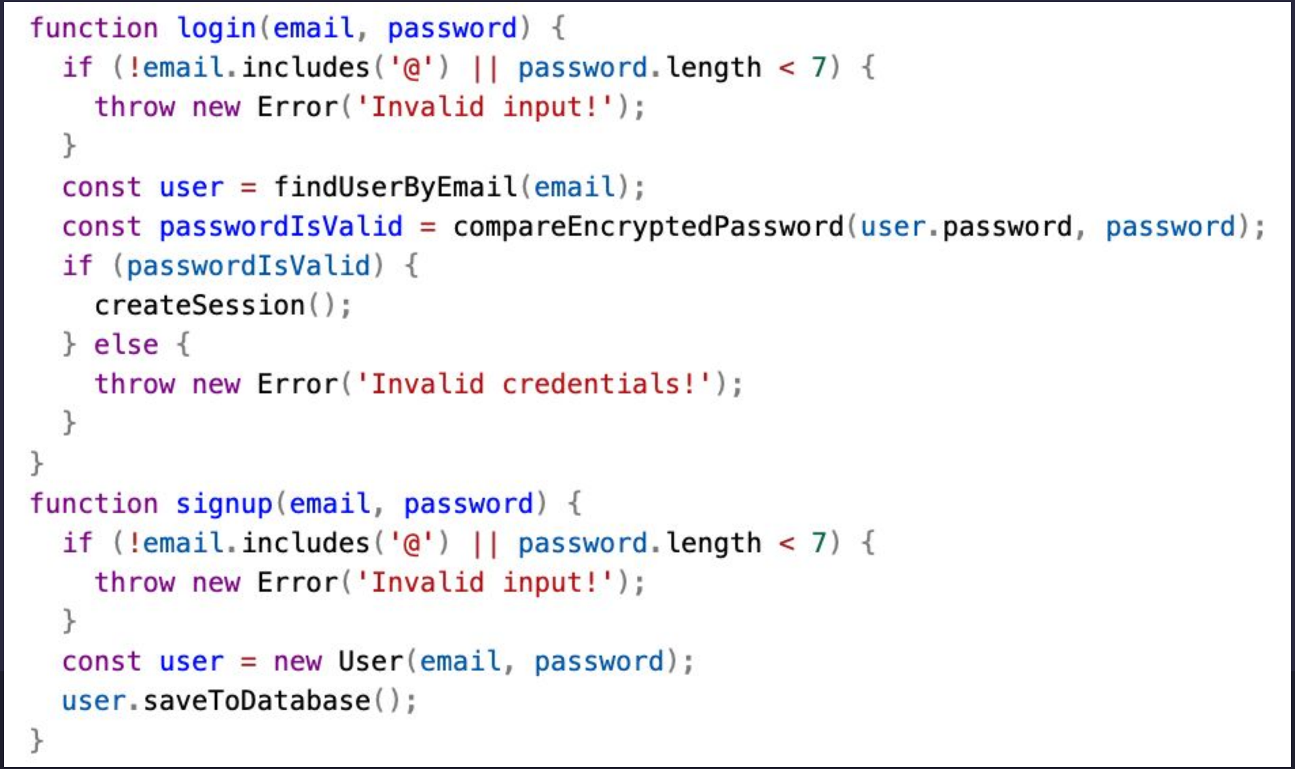
```
# Min. 8 characters, at least: one letter, one number, one special character
const passwordRegex = /^(?=.*[A-Za-z])(?=.*\d)(?=.*[@$!%*#?&])[A-Za-z\d@$!%*#?&]{8,}$/
```

3. Warnings - also in rare cases warning next to some code could make sense and improve readability - for example if a unit test may take a long time to complete or some functionality won’t work in certain environments

```
function fetchTestData() { ... } // requires local dev server
```

Comments & Formatting - Vertical Formatting

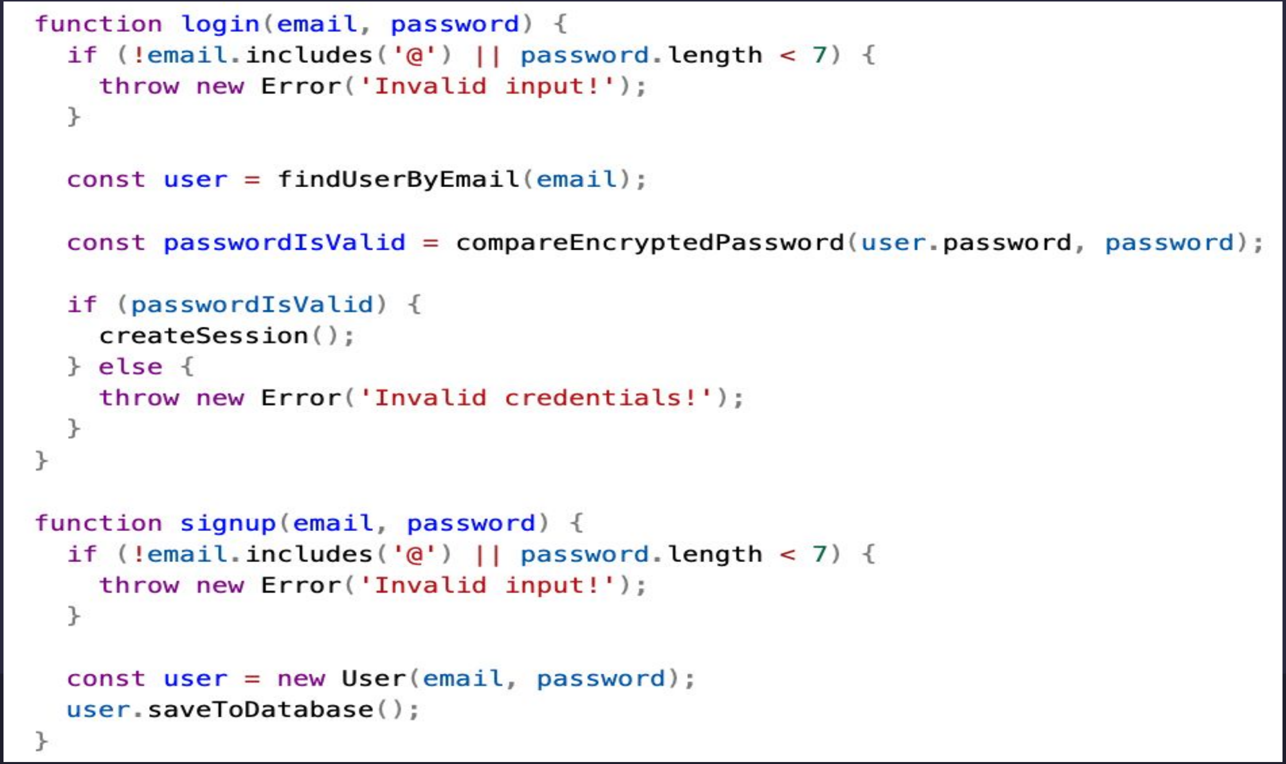
Vertical Formatting is all about using blank spaces where it'd be required



```
function login(email, password) {
  if (!email.includes('@') || password.length < 7) {
    throw new Error('Invalid input!');
  }
  const user = findUserByEmail(email);
  const passwordIsValid = compareEncryptedPassword(user.password, password);
  if (passwordIsValid) {
    createSession();
  } else {
    throw new Error('Invalid credentials!');
  }
}
function signup(email, password) {
  if (!email.includes('@') || password.length < 7) {
    throw new Error('Invalid input!');
  }
  const user = new User(email, password);
  user.saveToDatabase();
}
```

Comments & Formatting - Vertical Formatting

Vertical Formatting is all about using blank spaces where it'd be required



```javascript
function login(email, password) {
  if (!email.includes('@') || password.length < 7) {
    throw new Error('Invalid input!');
  }

  const user = findUserByEmail(email);

  const passwordIsValid = compareEncryptedPassword(user.password, password);

  if (passwordIsValid) {
    createSession();
  } else {
    throw new Error('Invalid credentials!');
  }
}

function signup(email, password) {
  if (!email.includes('@') || password.length < 7) {
    throw new Error('Invalid input!');
  }

  const user = new User(email, password);
  user.saveToDatabase();
}
```

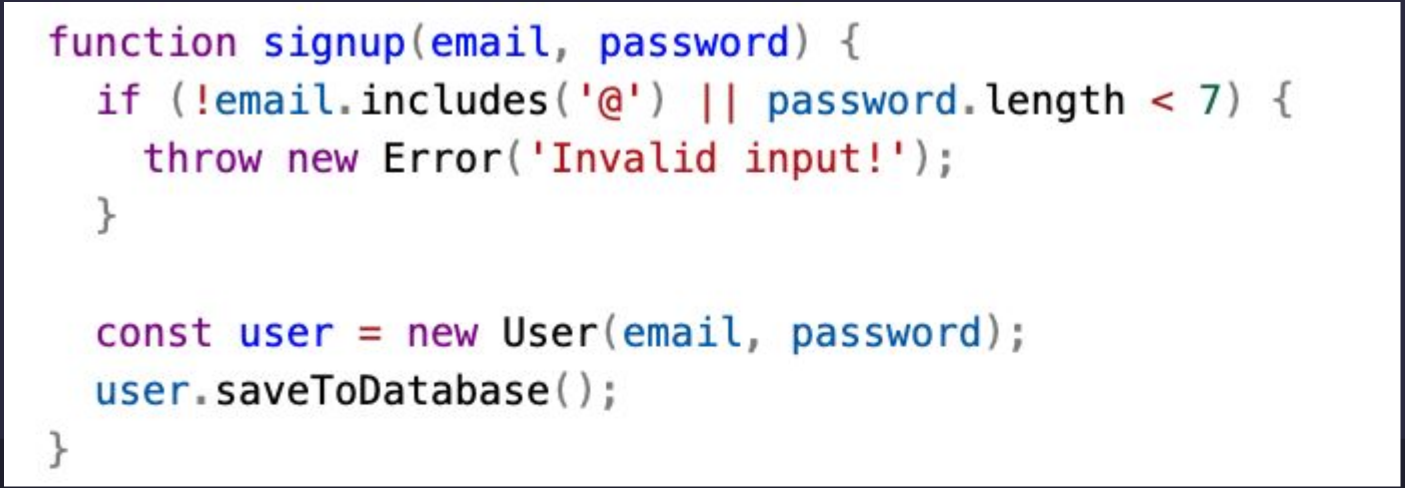
Comments & Formatting - Vertical Formatting

Vertical Density means that related concepts should be kept closely together

Vertical Distance means that concepts which are not closely related should be separated

Both concepts are used inside functions and outside of them.

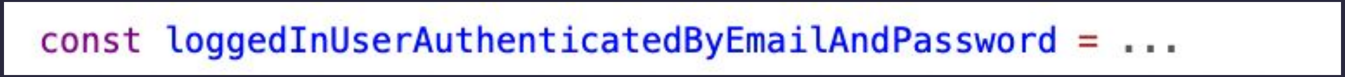
Example for a well placed vertical formatting -



```
function signup(email, password) {
  if (!email.includes('@') || password.length < 7) {
    throw new Error('Invalid input!');
  }

  const user = new User(email, password);
  user.saveToDatabase();
}
```

Comments & Formatting - Horizontal Formatting

Horizontal formatting is mainly about that lines should be kept short and readable.
This is mostly automatically done for us using our prettifier, but using short and precise naming methods would make the lines shorter and our code more readable -



```
const loggedInUserAuthenticatedByEmailAndPassword = ...
```

This variable name is way to specific, in contrast to **loggedInUser**

that would be more readable and take less time to read and understand.

Comments & formatting task

https://github.com/TzachiGitHub/cleanCodeWorkshop/blob/tasks/commentsAndFormattingTask.js

Functions

<u>Functions are made out of 3 parts:</u>

1. Their name
2. Their Parameters (if any)
3. Their body

<u>Functions parameters -</u>

The fewer parameters a function has, the easier it is to read and call it, and to understand the statement where it is being called.

```
logOut();    logIn(userName, password);     weather(city)
```

But what happens when we need more parameters?

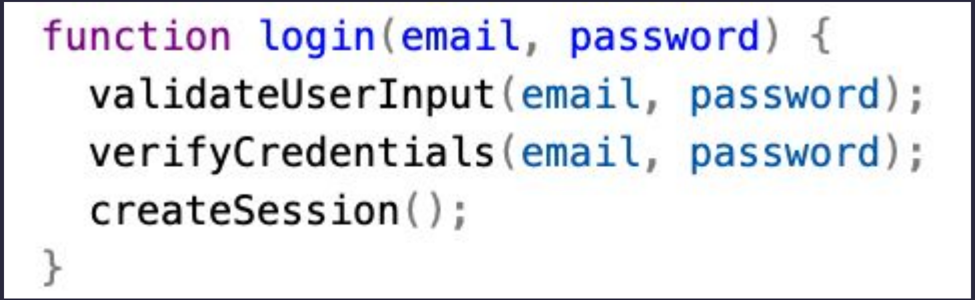
Functions parameters

We can’t always use functions without parameters or with just 1-2, so we should try to minimize their amount using arrays or more likely - maps:

```
createUser(“Nick”, “Fury”, “test@test.com”,  “Mrs. Cohen”, 31, [“Cleaning”, “Cooking”]);

Vs

Const userData = {
        userName: “Nick”,
        nickName: “Fury”,
        email: “test@test.com”,
        securityAnswer: “Mrs. Cohen”,
        idNumber: 31,
        hatesMost: [“Cleaning”, “Cooking”]
        };
createUser (userData);
```

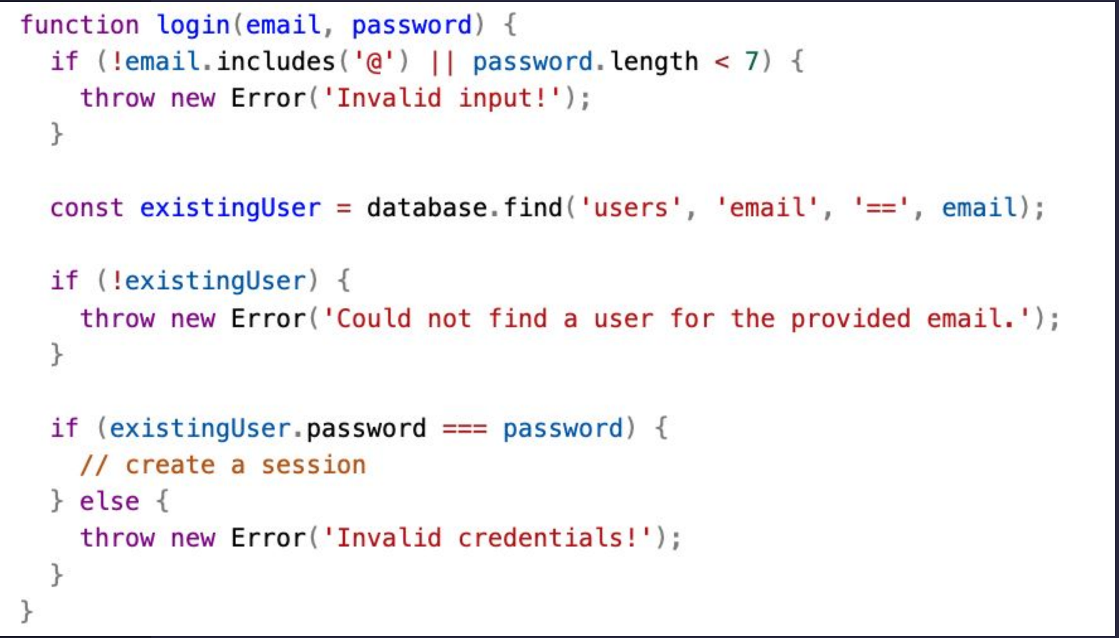
Function body - Levels of abstraction

In general functions are meant to do “one thing” -
Which is a simple way to say that it does one level of abstraction - but inside
it actually has lower levels of abstraction, which each of them are supposed to do
“one thing” in their level of abstraction. For example -

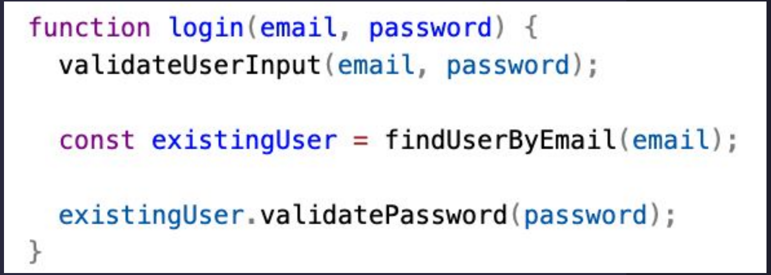


```
function login(email, password) {
  validateUserInput(email, password);
  verifyCredentials(email, password);
  createSession();
}
```

Function body - Levels of abstraction



```
function login(email, password) {
  if (!email.includes('@') || password.length < 7) {
    throw new Error('Invalid input!');
  }

  const existingUser = database.find('users', 'email', '==', email);

  if (!existingUser) {
    throw new Error('Could not find a user for the provided email.');
  }

  if (existingUser.password === password) {
    // create a session
  } else {
    throw new Error('Invalid credentials!');
  }
}
```



```
function login(email, password) {
  validateUserInput(email, password);

  const existingUser = findUserByEmail(email);

  existingUser.validatePassword(password);
}
```

Which way is easier to read & understand?

Function body - Levels of abstraction

Try to avoid mixing levels of abstractions -

```
function printDocument(documentPath) {
  const fsConfig = { mode: 'read', onError: 'retry' };
  const document = fileSystem.readFile(documentPath, fsConfig);
  const printer = new Printer('pdf');
  printer.print(document);
}
```

printDocument shouldn’t use know the settings for reading a file, but should just give the path to the file that needs to be read -

```
function printDocument(documentPath) {
  const document = readFromFile(documentPath);
  const printer = new Printer('pdf');
  printer.print(document);
}
```

And maybe even a bit more?

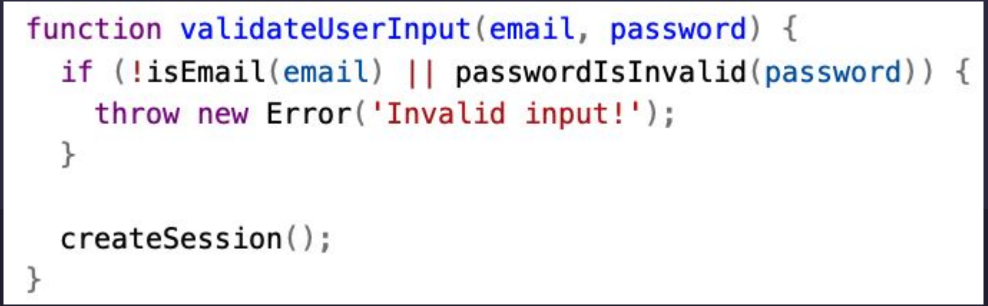
```
function printDocument(documentPath) {
  const document = readFromFile(documentPath);
  printFile(document);
}
```

Function body - Avoid Side Effects

Side effects happen when an operation, function or expression is modifying some state value outside its local environment, and except the value that function returns - you can see the observable effect of the function

Examples for side effects - connecting to a database, sending an HTTP request, Printing output, changing the state of global variables (useState, redux etc)

Side effects are perfectly normal thing in development, we do build applications in order to get results and change things -
The problems start when side effects are unexpected



```
function validateUserInput(email, password) {
  if (!isEmail(email) || passwordIsInvalid(password)) {
    throw new Error('Invalid input!');
  }

  createSession();
}
```

Does it make sense that
a validator creates a session?

Functions task

https://github.com/TzachiGitHub/cleanCodeWorkshop/blob/tasks/functionTask.js

Control Structures

No matter which code we’re writing - we’ll be using Control Structures - conditions and loops - ‘If’ statements, for/while loops, or ‘switch-case’ statements

and maintaining control structure is extremely important for the code flow, Because they can lead to bad or suboptimal code if used in a wrong way, or a clean and readable code if used correctly.

Hence we’ll try to focus on 3 main areas of improvements:
1. **Prefer positive checks**
2. **Avoid deep nesting**
3. **Embrace errors**

Control Structures - Prefer positive checks

It makes a lot more sense to use positive wording in your conditional checks instead of a negative wording -
For instance using our common front end **isFetching**
Is far more readable than naming it **isNotFetching**

```
{IsFetching &&
    <Item/>
 }

{!isFetching &&
    <DifferentItem />
 }
```

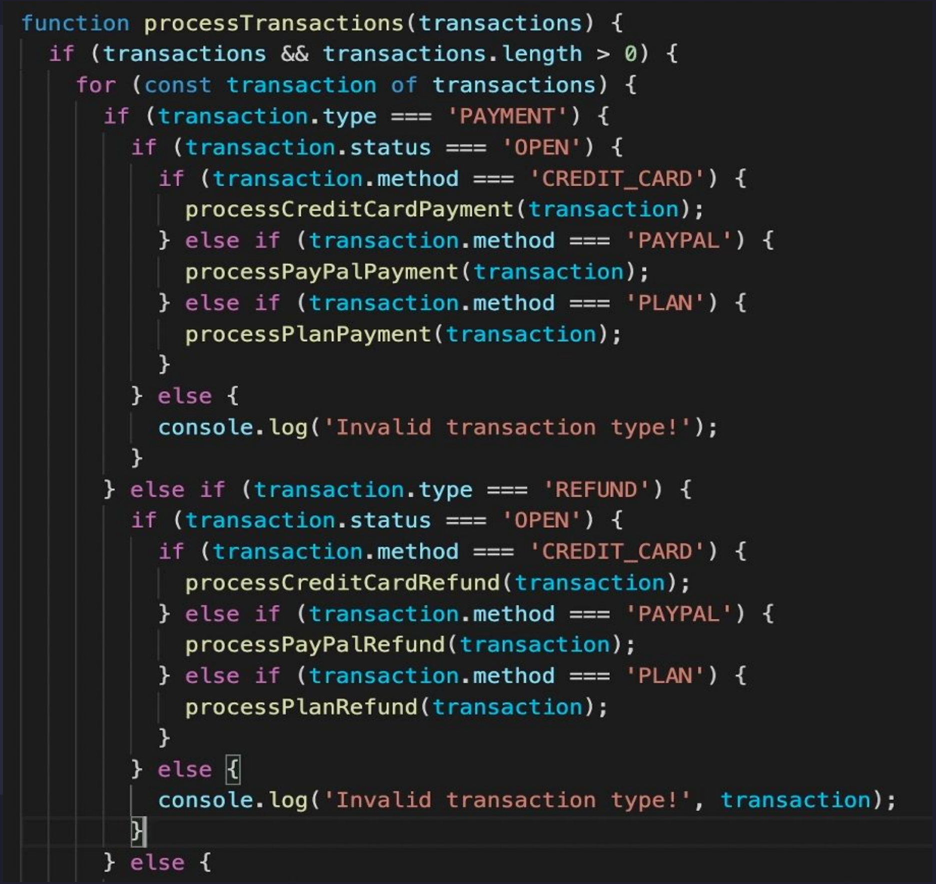
```
{IsNotFetching &&
   <Item/>
  }

{!isNotFetching &&
   <DifferentItem/>
  }
```

Control Structures - Avoid deep nesting

Having such code is highly unreadable, hard to maintain and also often error-prone

There are a few principles we'll focus on to prevent it:

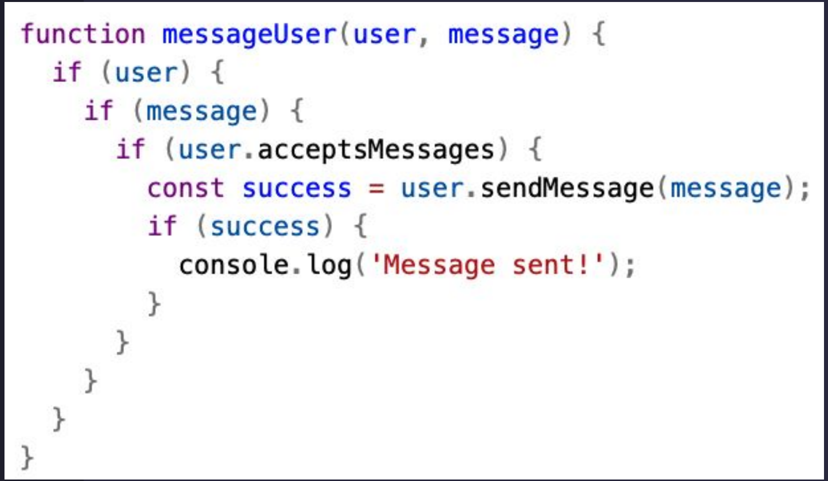
1. Use guards and fail fast
2. Extract control structures and logic into separate functions
3. Polymorphism & factory functions.
4. Replace 'if' checks with errors



```
function processTransactions(transactions) {
  if (transactions && transactions.length > 0) {
    for (const transaction of transactions) {
      if (transaction.type === 'PAYMENT') {
        if (transaction.status === 'OPEN') {
          if (transaction.method === 'CREDIT_CARD') {
            processCreditCardPayment(transaction);
          } else if (transaction.method === 'PAYPAL') {
            processPayPalPayment(transaction);
          } else if (transaction.method === 'PLAN') {
            processPlanPayment(transaction);
          }
        } else {
          console.log('Invalid transaction type!');
        }
      } else if (transaction.type === 'REFUND') {
        if (transaction.status === 'OPEN') {
          if (transaction.method === 'CREDIT_CARD') {
            processCreditCardRefund(transaction);
          } else if (transaction.method === 'PAYPAL') {
            processPayPalRefund(transaction);
          } else if (transaction.method === 'PLAN') {
            processPlanRefund(transaction);
          }
        } else {
          console.log('Invalid transaction type!', transaction);
        }
      } else {
```

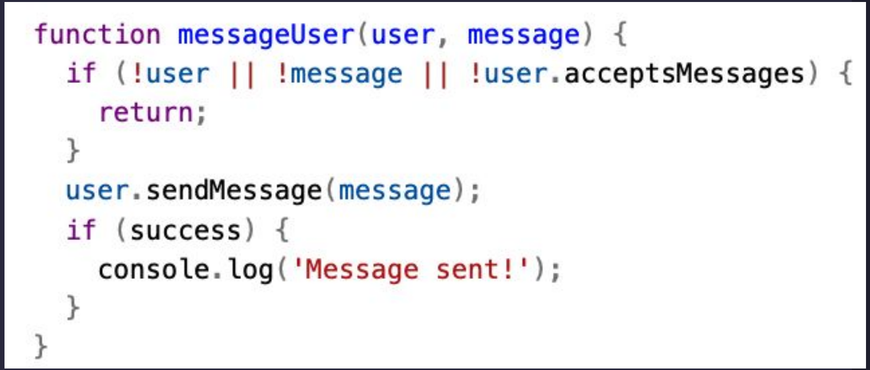
Avoid deep nesting - use guards and fail fast

Without guards:



```
function messageUser(user, message) {
  if (user) {
    if (message) {
      if (user.acceptsMessages) {
        const success = user.sendMessage(message);
        if (success) {
          console.log('Message sent!');
        }
      }
    }
  }
}
```

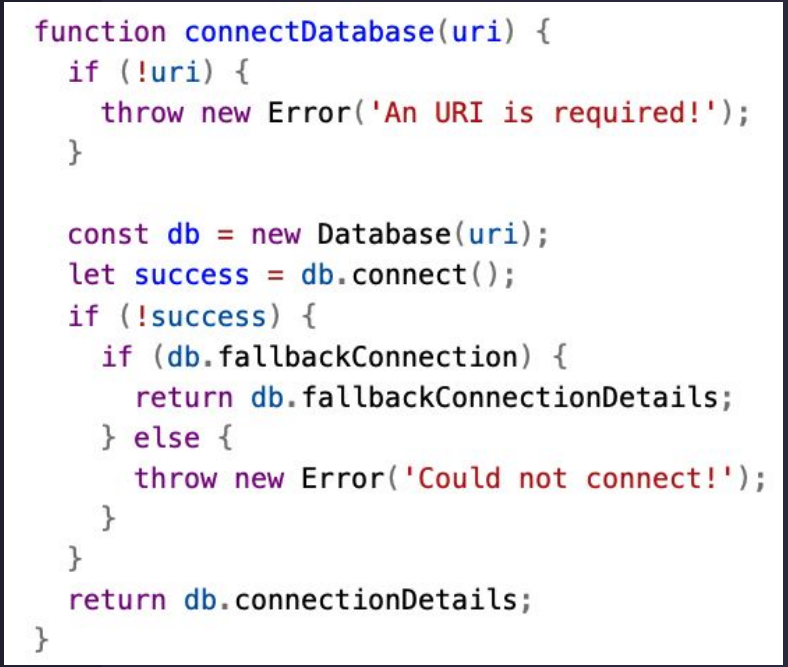
with guards:



```
function messageUser(user, message) {
  if (!user || !message || !user.acceptsMessages) {
    return;
  }
  user.sendMessage(message);
  if (success) {
    console.log('Message sent!');
  }
}
```

Avoid deep nesting - extract code to separate functions

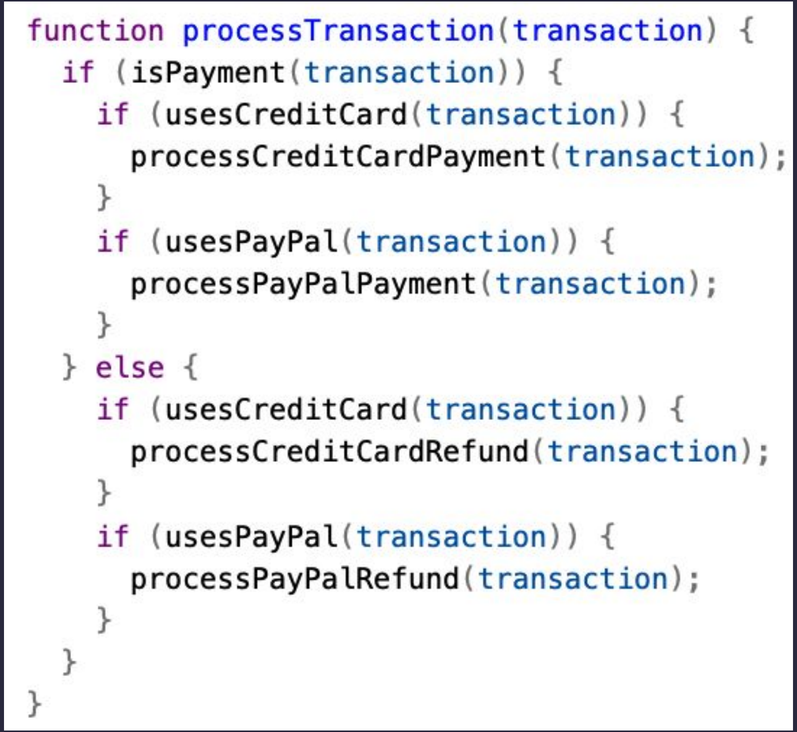
* Try to keep one level of abstraction difference - between the function name and body



```
function connectDatabase(uri) {
  if (!uri) {
    throw new Error('An URI is required!');
  }

  const db = new Database(uri);
  let success = db.connect();
  if (!success) {
    if (db.fallbackConnection) {
      return db.fallbackConnectionDetails;
    } else {
      throw new Error('Could not connect!');
    }
  }
  return db.connectionDetails;
}
```

```
function connectDatabase(uri) {
  validateUri(uri);

  const db = new Database(uri);
  let successfulConnection = db.connect();
  const  connectionDetails = successfulConnection ?
    db.connectionDetails : connectFallbackDatabase(db);

  return connectionDetails;
}
```

Avoid deep nesting - Polymorphism & Factory functions

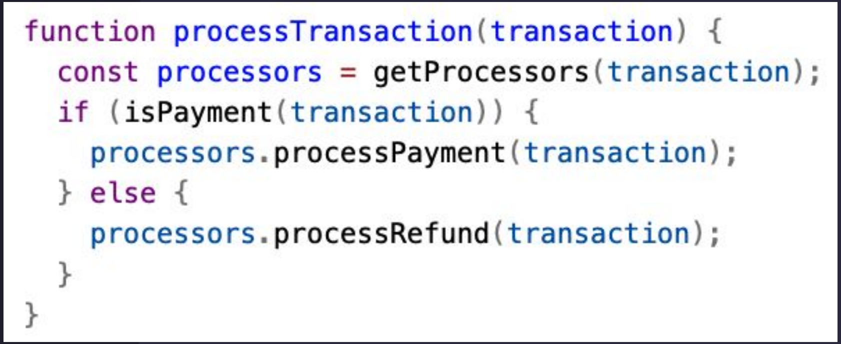


```
function processTransaction(transaction) {
  if (isPayment(transaction)) {
    if (usesCreditCard(transaction)) {
      processCreditCardPayment(transaction);
    }
    if (usesPayPal(transaction)) {
      processPayPalPayment(transaction);
    }
  } else {
    if (usesCreditCard(transaction)) {
      processCreditCardRefund(transaction);
    }
    if (usesPayPal(transaction)) {
      processPayPalRefund(transaction);
    }
  }
}
```

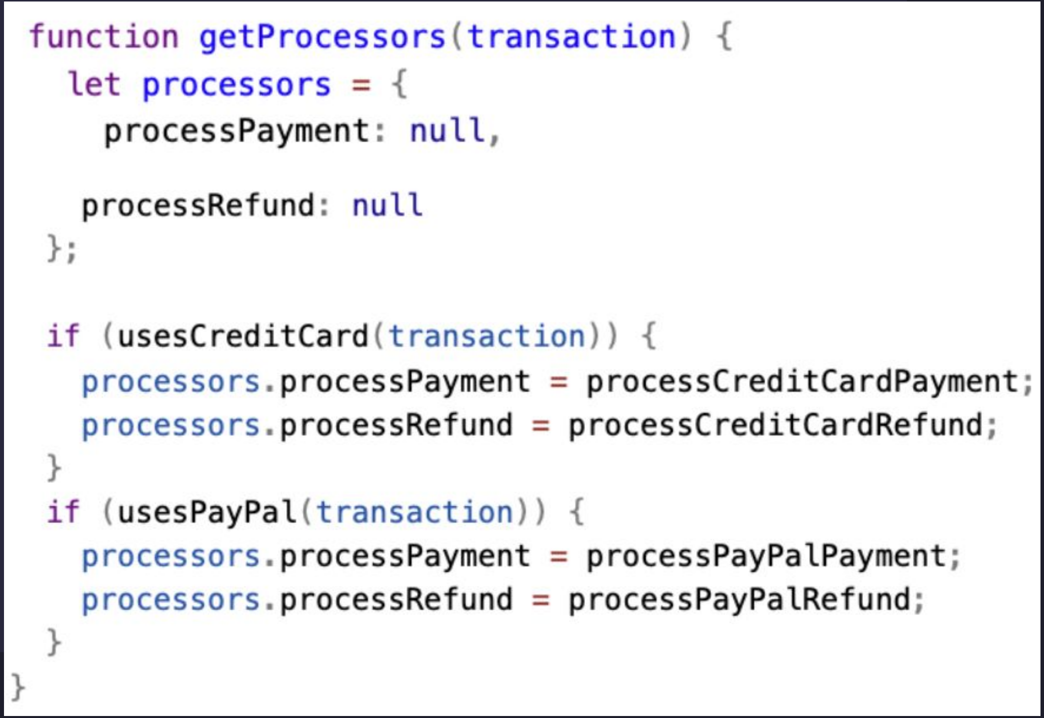
Before applying polymorphism and factory methods -
We can see that we repeat the use of usesCreditCard() & usesPayPal() because running different code depending on whether we have payment or refund

But what will happen if we'll use polymorphism & factory functions?

Avoid deep nesting - Polymorphism & Factory functions

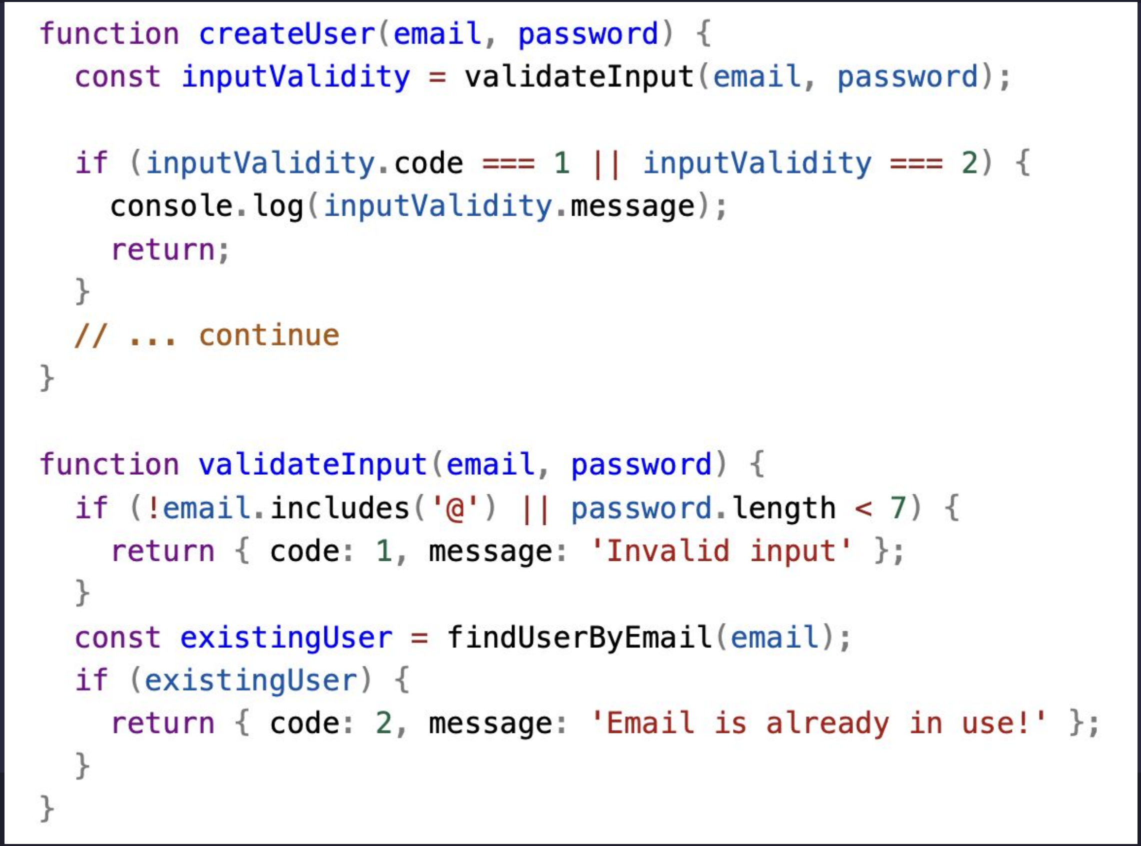


```
function processTransaction(transaction) {
  const processors = getProcessors(transaction);
  if (isPayment(transaction)) {
    processors.processPayment(transaction);
  } else {
    processors.processRefund(transaction);
  }
}
```



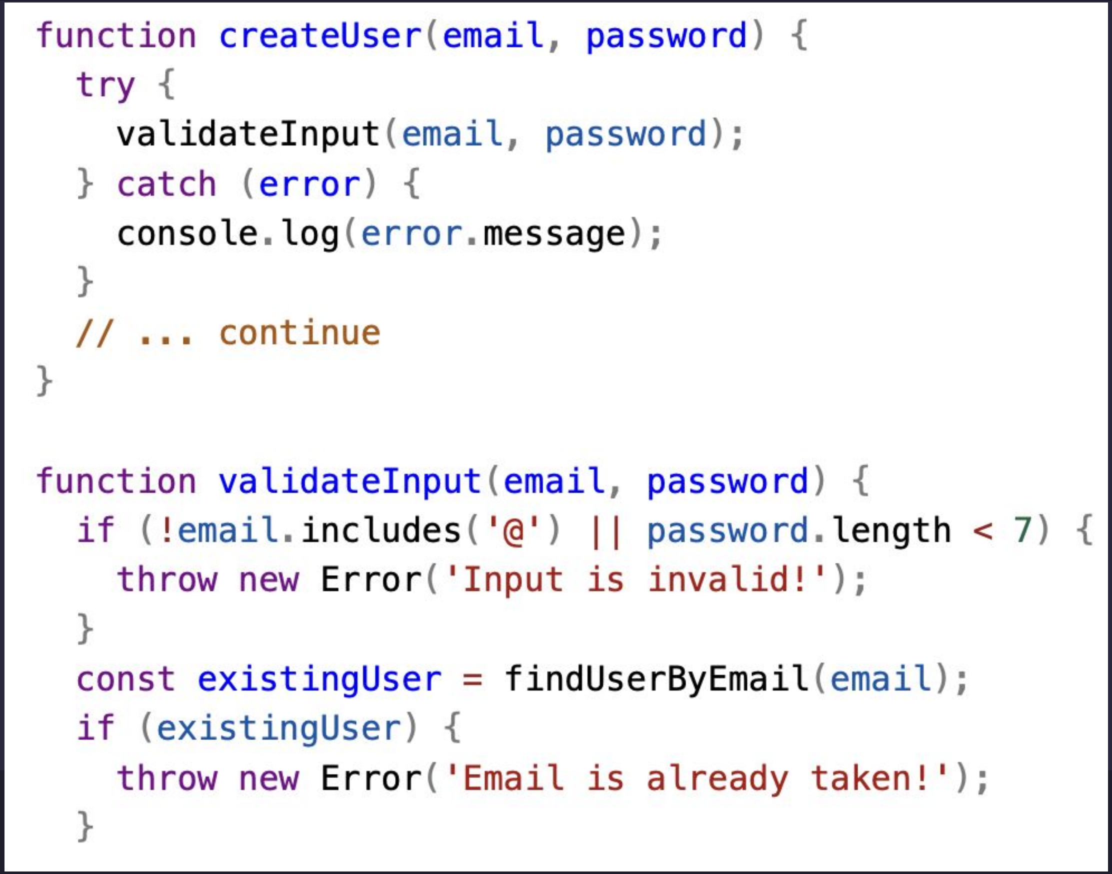
```
function getProcessors(transaction) {
   let processors = {
     processPayment: null,

    processRefund: null
  };

  if (usesCreditCard(transaction)) {
    processors.processPayment = processCreditCardPayment;
    processors.processRefund = processCreditCardRefund;
  }
  if (usesPayPal(transaction)) {
    processors.processPayment = processPayPalPayment;
    processors.processRefund = processPayPalRefund;
  }
}
```

Control Structures - Embrace errors



```javascript
function createUser(email, password) {
  const inputValidity = validateInput(email, password);

  if (inputValidity.code === 1 || inputValidity === 2) {
    console.log(inputValidity.message);
    return;
  }
  // ... continue
}

function validateInput(email, password) {
  if (!email.includes('@') || password.length < 7) {
    return { code: 1, message: 'Invalid input' };
  }
  const existingUser = findUserByEmail(email);
  if (existingUser) {
    return { code: 2, message: 'Email is already in use!' };
  }
}
```

Control Structures - Embrace errors



```
function createUser(email, password) {
  try {
    validateInput(email, password);
  } catch (error) {
    console.log(error.message);
  }
  // ... continue
}

function validateInput(email, password) {
  if (!email.includes('@') || password.length < 7) {
    throw new Error('Input is invalid!');
  }
  const existingUser = findUserByEmail(email);
  if (existingUser) {
    throw new Error('Email is already taken!');
  }
```

Control structure task

https://github.com/TzachiGitHub/cleanCodeWorkshop/blob/tasks/controlStructureTask.js

Naming

1. Use descriptive and meaningful names:
    * Variables and Properties: Nouns or short phrases with adjectives
    * Functions and Methods: Verbs or short phrases with adjectives
    * Classes/Components: Nouns
2. Be as specific as necessary and possible
3. Use yes/no “questions” for booleans (e.g. isValid, isInsured etc)
4. Avoid misleading names
5. Be consistent with your names (e.g. use ‘isFetching’ instead of ‘isGetting’)

Comments & Formatting

1. Most comments are bad - avoid them!

   Rare good comments might be -

   * Legal related
   * Warnings
   * Helpful explanations (clarifying something confusing like regex)
2. Use vertical formatting:
   * Keep related concepts close to each other (vertical density)
   * Add spacing/blank-lines between concepts that are not directly related (vertical distance)
3. Use horizontal formatting:
   * Avoid long lines - break them into multiple lines instead

Functions

1. Limit the number of parameters your functions use - less is better!
2. Consider using dictionaries/Arrays to group multiple parameters into one parameter.
3. Functions should be small and do one thing:
    * Level of abstraction inside the function body should be one level below the level implied by the function name.
    * Avoid mixing levels of abstractions in functions
    * Avoid redundant splitting
    * Stay DRY - Don’t repeat yourself
    * Avoid unexpected side effects

Control Structures & Errors

1. Prefer positive checks (isValid > isNotValid)

2. Avoid deep nesting:
    * Consider using “Guard” statement
    * consider using polymorphism and factory functions
    * Extract control structures into separate functions

3. Consider using real errors with error handling, instead of synthetic errors built with if statements

/THANKS!

/DO YOU HAVE ANY QUESTIONS?

YOUR LOGO HERE

/PROGRAMMING ICON PACK

INDEX.HTML